# Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory

Colin Blundell
University of Pennsylvania
blundell@cis.upenn.edu

Joe Devietti
University of Pennsylvania
devietti@cis.upenn.edu

E Christopher Lewis *
VMware, Inc.
lewis@vmware.com

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

## Abstract

Hardware transactional memory has great potential to simplify the creation of correct and efficient multithreaded programs, allowing programmers to exploit more effectively the soon-to-be-ubiquitous multi-core designs. Several recent proposals have extended the original bounded transactional memory to unbounded transactional memory, a crucial step toward transactions becoming a general-purpose primitive. Unfortunately, supporting the concurrent execution of an unbounded number of unbounded transactions is challenging, and as a result, many proposed implementations are complex.

This paper explores a different approach. First, we introduce the permissions-only cache to extend the bound at which transactions overflow to allow the fast, bounded case to be used as frequently as possible. Second, we propose ONETM to simplify the implementation of unbounded transactional memory by bounding the concurrency of transactions that overflow the cache. These mechanisms work synergistically to provide a simple and fast unbounded transactional memory system.

The permissions-only cache efficiently maintains the coherence permissions—but not data—for blocks read or written transactionally that have been evicted from the processor's caches. By holding coherence permissions for these blocks, the regular cache coherence protocol can be used to detect transactional conflicts using only a few bits of on-chip storage per overflowed cache block.

ONETM allows only one overflowed transaction at a time, relying on the permissions-only cache to ensure that overflow is infrequent. We present two implementations. In ONETM-Serialized, an overflowed transaction simply stalls all other threads in the application. In ONETM-Concurrent, non-overflowed transactions and non-transactional code can execute concurrently with the overflowed transaction, providing more concurrency while retaining ONETM's core simplifying assumption.

## Categories and Subject Descriptors

C.1.4 Computer Systems Organization [*Processor Architectures*]: Parallel Architectures

* This work was done while E Lewis was at the University of Pennsylvania.



*ISCA'07,* June 9–13, 2007, San Diego, California, USA.


## General Terms

Design, Languages, Performance

## Keywords

Transactions, Transactional Memory, Parallel Programming

## 1. Introduction

Transactional memory systems have been proposed to ameliorate the challenges of lock-based multithreaded programming [15]. Memory transactions are segments of code that execute atomically and in isolation, thus providing a synchronization mechanism similar to locks but with a more declarative interface. Fourteen years ago, Herlihy and Moss proposed an elegant and efficient hardware implementation of transactional memory [13]. This implementation allows transactions to execute concurrently, detecting conflicts at cache-block granularity and aborting transactions on conflicts to achieve isolation. Their implementation is simple in design as they leverage the existing on-chip caches and cache coherence protocol. Unfortunately, this implementation limits the volume of data that a transaction can access (to the size of a small on-chip cache) and does not allow transactions to survive a context switch, limiting the utility of transactions as a general-purpose synchronization primitive.

Recently, several proposals have emerged to provide unbounded hardware transactions by using the above implementation for transactions that do not overflow their on-chip resources or encounter an interrupt, while using a different hardware mechanism to handle the overflowed case (*e.g.*, [1, 7, 24]). These proposals support overflowed transactions with the same concurrency as non-overflowed transactions.

Unfortunately, these systems do not have simple implementations. As in the bounded case, they track the data that is associated with each transaction (for logging) and each memory block (for conflict detection). The size of the former depends on the number of memory references that can appear in a transaction, and the size of the latter can in the worst case be equal to the number of concurrently-executing transactions; in these systems, both quantities can be unbounded. To detect conflicts, a processor traverses the per-memory-block structure, and on a commit or abort, a processor traverses the per-transaction structure. Although various caching structures have been proposed to reduce the frequency of these traversals, the logic to maintain and manipulate them must still be implemented.

We take a different approach to unbounded hardware transactional memory. First, we propose a hardware mechanism, the *permissions-only cache*, that can be incorporated into many existing proposals to reduce the rate at which less-efficient overflow handling mechanisms are invoked. Second, because it is likely that

overflowed transactions will be rare, our proposal does not support unbounded concurrency among them, admitting a simpler implementation. Together, these two aspects of our proposal make the fast case common and the now-uncommon case simple.

To reduce the rate at which transactions overflow, we exploit the fact that the information necessary for performing conflict detection can be encoded in the coherence permissions of transactionally-accessed cache blocks; the data is not necessary. The permissions-only cache thus maintains only coherence permissions for transactionally-accessed blocks. The size reduction allows us to achieve a 256:1 compression ratio; *e.g.*, a 4-KB permissions-only cache can track up to a megabyte of transactionally-accessed data. Furthermore, dynamically allocating second-level cache frames for this purpose allows for transactions that have the potential to access hundreds of megabytes (as opposed to tens of kilobytes) without overflowing.

We also propose ONETM, a simple hardware-based approach for handling overflows by bounding concurrency among overflowed transactions. We explore two implementations. ONETM-Serialized revisits serialization as a viable approach for realizing overflowed transactions by stalling all other threads in an application when one transactions overflows. A similar mechanism was part of the original TCC proposal [12]. However, the concept has lost favor because (1) their implementation did not allow overflowed transactions to explicitly abort, and (2) serialization is unappealing when overflow is frequent. We present a serialization scheme that allows overflowed transactions to abort and relies on the permissions-only cache to ensure that overflow will be the uncommon case.

Although serialization may be sufficient if overflow is vanishingly rare, it may not be appropriate for all circumstances. ONETM-Concurrent provides more concurrency than ONETM-Serialized by allowing bounded transactions and non-transactional code to execute concurrently with a single overflowed transaction. The implementation for ONETM-Concurrent remains simple because it avoids the complex structures used to track an unbounded amount of state per memory block and retains simple commit and abort operations.

The next section describes a baseline bounded hardware transactional memory implementation. Section 3 introduces the permissions-only cache. Section 4 surveys some recent hardware-based unbounded transactional memory proposals. Section 5 presents ONETM. Section 6 experimentally evaluates ONETM, and Section 7 elaborates on additional related work.

## 2. Baseline Bounded Transactional Memory

This section describes the bounded hardware transactional memory system that we use as our baseline. The baseline implementation supports transactions that are bounded in terms of duration (*i.e.*, cannot survive context switches) and volume of data that may be accessed. This implementation of bounded transactions is similar to the original hardware transactional memory proposal [13] that many unbounded transactional memory proposals build upon.

Conflicts between transactions and other code are detected via the cache coherence protocol. Each block of each processor's private level-one data cache is augmented with two bits that record whether the block has been read or written within a transaction (called the *transactional read and write bits*, or just read and write bits for short). These bits interact with a standard invalidation-based cache coherence protocol to detect conflicts (*i.e.*, when two transactions have accessed the same memory block and at least one access is a write), which require one of the conflicting transactions to be aborted (or stalled [21]).

Our baseline system uses in-place updates and LogTM-style logging [21] to distinguish architected and speculative state (*i.e.*, state updated in a transaction).[1] Before a speculative store updates memory, the block's address and old value are written to a transaction-private log mapped into the thread's virtual address space. To avoid logging the same memory block multiple times, log updates are elided when the write bit associated with the block is already set, indicating that it has previously been logged.

In-place updates allow efficient commits. When a transaction commits, the read and write bits in the cache are simply flash cleared (committing the speculative data to the state of the machine), and the log is discarded by restoring the log pointer to the beginning of the log buffer. When a transaction aborts, the system iterates over the log entries in hardware or software, restoring each block. After the log has been restored, the processor flash clears the in-cache read and write bits.

This implementation supports only bounded transactions because the cache-based mechanism for conflict detection limits both the volume of data that may be accessed within a transaction and the duration of a transaction. Data access volume is limited because cache lines cannot be evicted (otherwise read and write bits would be lost, preventing the detection of subsequent conflicts with the evicted data). Duration is limited because the in-cache read and write bits implicitly belong to the currently executing transaction. This implementation has no mechanism for transferring read and write bits from the cache to architected state, and so it must abort transactions on a context switch. Note that the logging mechanism bounds neither aspect of transaction execution.

Recent work has sought to support hardware-based transactional execution that imposes no spatial or temporal bounds [15]. For blocks that do not overflow on-chip state, many of these proposals use the efficient mechanism described above; for blocks that overflow on-chip state, such proposals use other higher-overhead mechanisms (see Section 4 for examples). In the next section, we describe a mechanism that efficiently tracks a much larger amount of conflict detection information on-chip, helping to ensure that these overflow mechanisms are rarely needed.

## 3. Making the Fast Case Common

In this section we introduce a mechanism, the *permissions-only cache*, that reduces the frequency with which transactions overflow on-chip resources. The permissions-only cache allows a transaction to access more data than can be buffered on-chip *without* transitioning to a higher-overhead overflowed execution mode. The permissions-only cache efficiently tracks transactional read and write bits for blocks that have been replaced from the processor's data cache by retaining coherence permissions—but not data—for these blocks. Only when the permissions-only cache itself overflows does the system need to fall back on some other mechanism for detecting conflicts for overflowed blocks. Overflowed blocks can then be managed using one of the many previously proposed schemes (*e.g.*, [1, 7, 8, 10, 14, 21, 24]) or one of the implementations we propose in Section 5.

### 3.1 Operation

The permissions-only cache tracks conflict information for blocks that have exceeded the capacity of the data cache. It is organized as a tagged, set-associative structure that contains a read bit and a write bit per entry. Its efficient data-less encoding of coherence permissions is similar to the Store Miss Accelerator [6], which retains

[1] In contrast, the original hardware transactional memory proposal buffered speculative state in a special cache [13].

exclusive coherence permission to evicted blocks. The permissions-only cache is (1) read by external coherence requests as part of conflict detection, (2) updated when a transactional block is replaced from the data cache, (3) invalidated on a commit or abort, and (4) read on transactional store misses to avoid redundantly logging the block (*i.e.*, if the write bit was already set in the permissions-only cache, logging would be redundant). Thus, the permissions-only cache is off the critical path of accessing the first-level data cache. Moreover, the permissions-only cache need not be accessed at all for processor-local memory operations.

Just as external coherence requests check the read and write bits in the data cache to detect conflicts, external requests also check the bits in the permissions-only cache (performed in parallel with the data cache lookup). When a block's write bit is set in the permissions-only cache, the local coherence state of the block is *clean-exclusive without data*; when only the read bit is set, the state is *shared without data*. Externally, these states are indistinguishable from the traditional *clean-exclusive* (*E*) and *shared* (*S*) coherence states. Existing protocols commonly allow silent replacement of *S* and *E* blocks, and thus already implicitly support these new states.

When a transactional block is replaced from the data cache, the processor sets the appropriate read/write bits in the permissions-only cache, allocating an entry if necessary. If the replaced data cache block was dirty (modified), the block is written back to the second-level cache or memory; non-dirty (clean) blocks are silently discarded. Dirty blocks may safely escape because any remote read to these addresses will conflict with the transactional write bit in the permissions-only cache, preventing any access to the block until the subsequent abort has successfully restored the pre-transactional value. On a transaction commit or abort, the permissions-only cache is cleared by flash-invalidating all its blocks. The permissions-only cache is often empty (and thus need not be searched); in these circumstances, it can be completely powered down to save dynamic and static power.

## 3.2 Efficient Encoding

Because the permissions-only cache does not contain data, it can more efficiently encode the transactional read/write bits (just a few bits per block) than other on-chip caches that hold data as well as addresses. A naive implementation of a permissions-only cache would incur the overhead of a full cache tag for each two-bit entry. However, by using sector cache techniques [16] this tag overhead can be reduced dramatically. A 512-bit (*i.e.*, 64-byte) entry per tag would provide for 256 two-bit sectors (containing a read bit and a write bit). A single 256-sector entry maps a contiguous 16KB region of memory (256 sectors $\times$ 64B cache lines), a 256-to-1 compression ratio in the best case. In the worst case of poor spatial locality, a 4KB permissions-only cache would track the read/write bits for only 4KBs of blocks. However, with good page-level spatial locality, a 4KB permissions-only cache allows a transaction to access up to 1MB of data without overflow. Figure 1 shows the indexing and organization of a 4KB 256-sector direct-mapped permissions-only cache.

To support even larger transactions without overflow, instead of using a dedicated structure, the processor can dynamically share the second-level cache's storage capacity by allowing second-level cache frames to contain either data *or* an array of read/write bits. A second valid bit—a *permissions-only valid bit*—is added to each entry's cache tag to indicate when the frame holds transactional read/write bits. When a transactional block is replaced from the data cache, its transactional read/write bits are updated in the corresponding bits in the second-level cache's data array (replacing and allocating another entry as needed). On a commit or abort of

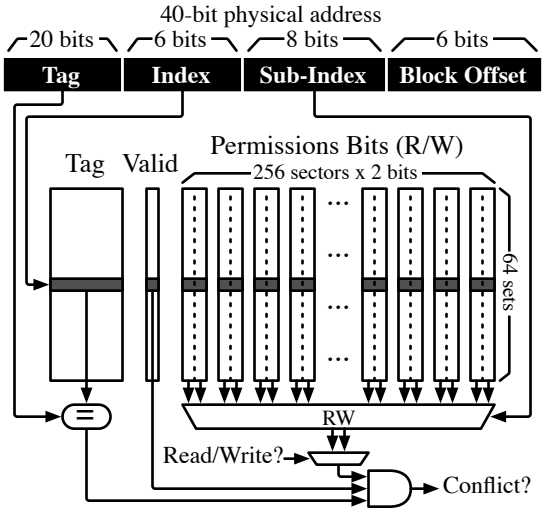


**Figure 1. A 4KB 256-sector permissions-only cache.**

a transaction, all the read/write bits are discarded by flash-clearing the permissions-only valid bits. For shared second-level caches, an additional core identifier field is associated with each cache frame containing permissions data.

When external coherence invalidations query the second-level cache, the cache tags are accessed twice. The first tag lookup is the normal lookup, but it will match only for frames that hold data blocks (by checking the permissions-only valid bit). The second tag lookup—which uses the sector cache indexing similar to a stand-alone permissions-only cache—checks for matching frames of read/write bits. When a tag hit occurs on a tag for a frame of read/write bits, the data array is accessed to query the corresponding bit (sector) to detect conflicts. If no permissions-only blocks have been allocated in the second-level cache, the second lookup is skipped.

With such an organization, a 4MB second-level cache with 64-byte blocks can hold enough permissions-only information to allow a transaction to access up to 1GB of data (64K entries of 256 read/write bit pairs and each entry maps 16KBs) without overflow.

## 3.3 Discussion

By efficiently tracking transactions' read and write sets, the permissions-only cache increases the size of transactions that can successfully complete without invoking an overflowed execution mode, largely independent of the particular scheme used to handle overflows. A permissions-only cache would likely provide little benefit to LogTM [21], which uses a sticky state at the memory to support lightweight gradual cache overflow. However, the reduction in the frequency of overflows may reduce the runtime overheads of other previously proposed hardware-based unbounded transactional memory schemes (*e.g.*, [1, 24]). A permissions-only cache could also extend the in-cache tracking of blocks used by hardware-assisted software transactional memory proposals (*e.g.*, [25, 26]) or hybrid hardware/software transactional memory proposals (*e.g.*, [8, 10, 14]).

With a sufficiently large permissions-only cache, the occurrence of overflowed transactions will likely be rare. In Section 5 we explore an approach for handling overflows that is enabled by this assumption. Before describing our proposed techniques, we review how some prior approaches for hardware-based unbounded transactional memory handle overflows.

# 4. Background on Unbounded Hardware TM

To provide basis for later comparison, this section describes three hardware-based unbounded transactional memory proposals that precisely detect conflicts at the cache block granularity: UTM [1], VTM [24], and PTM [7]. We defer discussion of XTM [8], a hardware/software proposal that uses the virtual memory system to manage overflows, as well as LogTM-SE [31] and Bulk [5], two signature-based proposals, until Section 7.

## 4.1 UTM

UTM [1] was the first transactional memory proposal to support unbounded transactions (in both size and duration). UTM maintains its transactional state in a single, shared, memory-resident data structure called the xstate. The xstate structure contains (1) logs for each active transaction to record read and written addresses and the original data values at the written addresses, and (2) for each block in memory, a read/write bit and a linked list of pointers into the log entries for that block. New versions of transactionally-written blocks are stored in-place (*i.e.*, UTM uses eager version management [21]). On an overflow, a processor adds an entry to its log, optionally walking the list of entries for the overflowing block to avoid redundant logging. Conflicts are detected by first inspecting the RW bit; if that bit signals a conflict, the transaction walks the linked list associated with the block to determine whether other transactions have accessed the block (*i.e.*, whether there is actually a conflict). An aborting transaction walks its list of accesses, destroying the log and reverting memory state. A committing transaction traverses the list of accesses to clean up the log. The xstate structure may be concurrently updated and read by multiple threads.

## 4.2 VTM

VTM [24] tracks overflowed transactional state using a shared data structure mapped into the virtual address space (called the XADT). Entries in the XADT are allocated when blocks overflow the cache. Much like UTM's xstate, VTM's XADT uses linked lists and supports accessing all entries for a specific virtual memory block or all entries for a specific transaction. XADT operations include concurrently adding an entry on overflow, looking up an entry for a block, committing a transaction, aborting a transaction, and saving state on context switches. Each transactional load or store miss checks for conflicting transactional accesses before it completes. Unlike UTM, VTM buffers speculative updates in the XADT itself, propagating these updates only when a transaction commits (*i.e.*, VTM uses lazy version management [21]).

To reduce expensive walks of the XADT, VTM introduces two caching mechanisms. First, VTM introduces a counting-Bloom-filter-based table (the XF) accessed on cache misses to quickly rule out conflicts with other transactions. Only when the XF indicates a potential conflict must the processor walk the XADT. The XF is mapped into the virtual memory space, shared among all threads, and accessed with cacheable loads and stores; as such, the XF can create overheads due to coherence sharing misses. Second, VTM employs another table, the XADC, to cache XADT entries for blocks that have been accessed by the current transaction.

On commit, VTM walks all the XADT entries for the transaction, copies the non-committed values into the memory, updates the shared XF, and deallocates and unlinks the XADT entries. Although non-transactional loads and stores do not normally need to check the XADT/XF, they must do so when a transaction is committing. An abort similarly walks the list of entries for the aborting transaction; this walk can be done in the background.

On a context switch, VTM walks the cache and overflows any transactionally read or written blocks. As updating the XADT requires virtual addresses and most caches are physically tagged, VTM's cache is augmented with virtual address tags. When a transaction is swapped back in after a context switch, all of the values read by that transaction are validated by comparing the current value of the block with the value previously read by the transaction for the block, requiring the buffering of both reads and writes.

## 4.3 PTM

PTM [7] supports unbounded transactional memory by associating state with physical addresses at the block granularity but allocating/reallocating this shadow state on a per-page basis. PTM's shadow pages behave similarly to UTM's log pointers, except PTM's Transaction Access Vector (TAV) lists track data for an entire page. Like both UTM and VTM, the transactional state data structure supports iterating over all entries associated with both a particular memory location and a particular transaction. PTM simplifies data logging by making the observation that because only one transaction can be writing a block at a time (because of its use of eager conflict detection), one shadow copy for each block is the maximum ever needed.

In PTM, all of the transactional state is maintained and accessed at the memory controller during cache misses. The memory controller is responsible for all conflict detection, updating transactional state, and aborting/committing transactions. To avoid performing a list walk of TAVs on each cache miss, PTM employs a TAV summary cache at the memory controller (different from, but analogous to VTM's XF). When a cache block overflows the cache, it is the memory controller that is responsible for recording the original and overflowed value.

On commit, the memory controller walks and updates all of the TAVs for the transaction and updates the summary vectors. Abort is similar, but the controller copy-restores the original values. The proposal also describes an optimized version in which non-speculative blocks can reside in either the home or shadow page, with a bit vector specifying which page has the non-speculative copy of each block. In this version, the memory controller toggles the bits for transactionally-accessed blocks on commit but does not have to copy-restore blocks on abort (it still walks the TAV list for the transaction to free its entries, however). Between the time a transaction logically commits and completes clearing its transactional state, the transaction is marked as committed, signaling that conflicts due to the committing transaction can be ignored.

To avoid overflowing all transactional blocks on a context switch, PTM associates a transaction identifier with each block in the cache. However, the PTM proposal assumes that the in-cache transaction identifiers are cleared in the case when a transaction resumes execution and commits on another processor, but does not explain how that is done.

## 4.4 Discussion

These three proposals require the hardware to dynamically allocate/deallocate, maintain, and concurrently manipulate complex link-based structures (UTM's xstate, VTM's XADT, and PTM's Transaction Access Vectors) and the corresponding cached versions of these structures. Manipulating and accessing these structures can add overhead to both overflowed transactions and concurrently-executing non-overflowed transactions, which need to access these structures to perform conflict detection. More importantly, the hardware for correctly manipulating these structures is not simple.

In the previous section, we described how the permissions-only cache can be employed to optimize the performance of these proposals. However, the knowledge that overflows will likely be rare

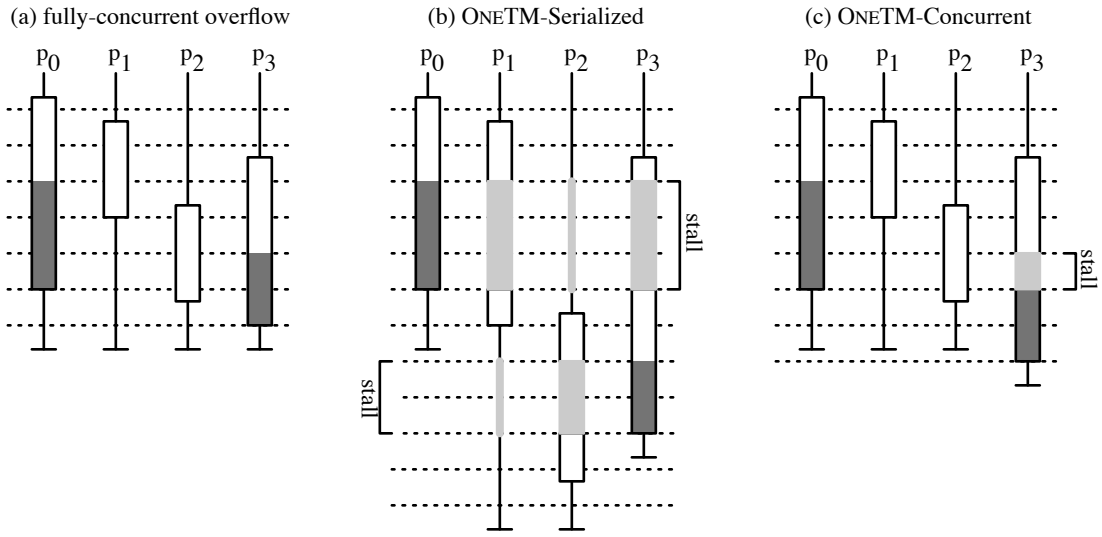


**Figure 2. An example execution on three systems handling overflowed transactions in different ways. The white bars represent non-overflowed transactions, the dark gray bars overflowed transactions, and the straight lines non-transactional execution. A light gray color means that the processor is stalled. In this execution, there are no conflicts, and the same amount of useful work is performed on each system.**

also allows us to consider new design points for handling overflows that trade off performance for simplicity. In the next section, we describe two such systems built around the idea of bounding concurrency of overflowed transactions.

## 5. Making the Uncommon Case Simple

We propose ONETM, a transactional memory system in which only a single overflowed transaction per process can be active at a time. The principal advantage of this design is that the implementation is relatively simple. The impact of the concurrency restrictions on overall system throughput is small because the permissions-only cache ensures that overflows will be the uncommon case.

This section describes two instantiations of ONETM. In ONETM-Serialized, overflowed transactions serialize the system (while retaining the ability to abort). In ONETM-Concurrent, any number of non-overflowed transactions (as well as non-transactional code) are permitted to execute concurrently with the single overflowed transaction. In Figure 2, we illustrate the differences in concurrency between these implementations as well as a system that places no concurrency restrictions on overflowed transactions (such as the systems described in the previous section).

### 5.1 ONETM-Serialized

Our first implementation revisits the idea of serialization of overflowed transactions first proposed in TCC [11]. Our serialized implementation simply stalls all other threads in an application when one of the threads needs to execute an overflowed transaction, as illustrated in Figure 2b (threads executing non-transactionally must stall to maintain strong atomicity [2]). Unlike the original TCC proposal, however, overflowed transactions still support an explicit abort operation because overflowed transactions continue to log.

To transition to overflowed execution, the processor must ensure that no other thread in the application is executing in overflowed mode. Exclusivity of overflowed execution is achieved via the *shared (per-process) transaction status word* (STSW), which resides in a fixed location in the virtual address space of each process. The STSW contains an *overflowed* bit, which is set while any thread in the application is executing an overflowed transaction. This bit acts much like a mutex lock on overflowed execution. A transaction may only transition to overflowed execution after it has atomically changed the bit from unset to set. If it finds the bit set, the processor must stall (*i.e.*, spin until the overflow bit is unset) or alert the operating system to schedule another thread/process. Finally, when an overflowed transaction commits, it unsets the overflowed bit in the STSW. The fields of the STSW are summarized in Figure 3a; the OTID field will be introduced in Section 5.2.2.

To serialize execution during overflow, all threads in the application that are not executing an overflowed transaction must monitor the overflowed bit in the STSW and stall if it is set; they resume only when the bit is unset (as illustrated in Figure 2b). The STSW can be coherently cached in a special register to make these checks inexpensive. Conflicts between the overflowed transaction and another (stalling) thread are resolved in favor of the overflowed transaction; if an overflowed transaction is context-switched out, the other threads continue to stall on the overflowed bit in the STSW. STSW accesses are not part of a transaction's read or write sets, so updates to the STSW never cause conflicts.

In addition to the STSW, each thread in the system has a *private (per-thread) transaction status word* (PTSW). The PTSW is an architected machine register (*i.e.*, it persists across context switches because the operating system saves and restores this register along with all the other architected registers). The PTSW contains an *overflowed* bit (set only when the *current* thread is executing an overflowed transaction) and a *transaction nesting depth* (TND) field. The TND field is used to implement the subsumption of nested transactions (*i.e.*, nested transaction initiation and commit are treated as no-ops, except for the manipulation of the TND field). Because the PTSW persists across context switches and migrations, a thread will not forget that it is executing an overflowed transaction nor the nesting depth of the current transaction. The fields of the PTSW are summarized in Figure 3b; the *no-user-abort* field will be introduced in Section 5.5.

Although this implementation is simple, the price of its simplicity is the loss of all concurrency when a transaction overflows,

which will have a significant negative impact on performance if overflows are frequent. Although the addition of the permissions-only cache can substantially reduce overflows (and thus the performance ramifications of serializing on overflow), we next propose an implementation that allows concurrent execution of non-overflowed transactions, non-transactional code, and a single overflowed transaction while remaining relatively simple.

## 5.2 ONETM-Concurrent

In order to allow other code to execute concurrently with a single overflowed transaction (Figure 2c), we introduce per-block persistent transaction metadata as part of the architected state. The system uses this metadata to track the read and write set of the single overflowed transaction; other threads then check the metadata to detect conflicts. To efficiently provide this metadata, each cache-block-sized block of physical memory is augmented with additional bits; these bits are the overflowed equivalents of the read/write bits described in Section 2. When the overflowed transaction writes (reads) a block, it sets the overflowed metadata write (read) bit. Non-overflowed transactional and non-transactional accesses detect conflicts with the overflowed transaction by inspecting these bits. A single set of bits per memory block is sufficient because there can be only one overflowed transaction at a time.

We next describe the use and manipulation of the metadata, followed by a description of how it can be efficiently set and cleared.

### 5.2.1 Metadata Operation

The additional storage for the metadata is implemented in the memory controllers. Each memory controller allocates a fixed-sized region in its physical memory to store the metadata associated with its remaining addressable memory. If we allocate two bytes (two bits indicating transaction read and write and a 14-bit identifier to be described later) for each 64-byte block, this represents only a 3% memory overhead.

The per-block metadata is part of the system's architected state, existing both in caches (in addition to transactional read/write bits used by non-overflowed transactions) and memory. As the metadata is logically associated with every block of data, the metadata travels with the data anytime the data block is transferred (*e.g.*, cache misses, responses from memory, cache-to-cache data transfers, and cache evictions). When responding to a cache miss, the memory controller provides both the data and metadata bits from the memory in parallel.[2] Although this metadata increases the size of the data payload, the coherence protocol itself need not change, and thus no special logic is required to communicate and manage the metadata. Non-overflowed transactions check for conflicts by simply examining the overflowed metadata of a cache block after they have brought the block into their cache.

When a transaction overflows, it transitions to overflowed execution mode. A simple way to accomplish this transition is to abort the transaction and restart it in overflowed mode after ensuring that no other thread in the application is already executing in overflowed mode (as described in Section 5.1). This is the specific implementation that we evaluate in Section 6. Alternatively, ONETM-Concurrent can avoid an abort by more gracefully transitioning to overflowed mode. As before, the processor must first ensure that no other thread in the application is executing in overflowed mode. Next, the processor walks both the data cache and the permissions-only cache to set the overflowed metadata for blocks read or written by the transaction; this action ensures that the conflict detection information for these blocks is not lost if the overflowed transaction is context switched. As a further optimization, the processor could update the metadata gradually as blocks overflow the caches and defer the metadata updates for non-overflowed blocks until a context switch actually occurs.

[2] As the memory access path is wider than two bytes, the memory controller caches the rest of the data read, reducing memory traffic by exploiting spatial locality in the reference stream.

**(a) Shared Transaction Status Word (STSW)**

Located at a fixed address in virtual memory known to all threads

| STSW Field | Description |
|---|---|
| overflowed? | is there a current overflowed transaction? |
| OTID | ID of current overflowed transaction |

**(b) Private Transaction Status Word (PTSW)**

Per-thread architected register, saved/restored on context switches

| PTSW Field | Description |
|---|---|
| overflowed? | is this thread in an overflowed transaction? |
| TND | nesting depth of current transaction |
| no-user-abort? | disables logging, allows IO |

**Figure 3. Description of transaction status words.**

### 5.2.2 Lazy Metadata Clearing

When an overflowed transaction commits or aborts, we would conceptually like to clear all metadata that the transaction has set. However, the number of blocks with non-zero overflowed transactional metadata is unbounded, and such blocks could be in any cache, memory module, or even swapped to disk. As such, it is not possible to easily clear all the overflowed transactional metadata.

Instead of actively clearing the metadata, the system clears the metadata lazily by using an *overflowed transaction identifier* (OTID) to differentiate between stale and current metadata. The per-block metadata is extended to hold an OTID (the 14-bit identifier mentioned earlier) that is updated anytime the metadata read/write bits are set. The OTID of the active overflowed transaction is also stored in the STSW (see Figure 3a), allowing all processors to fetch the current OTID by executing a coherent read request to its location. When a transaction transitions to overflowed mode, it increments the OTID in the STSW. Instead of explicitly clearing the metadata bits when it completes, the overflowed transaction simply clears the overflowed bit in the STSW as before.

A processor checks for conflicts by checking the metadata as described above; the processor elides this check if the overflowed bit in the PTSW is set or the overflowed bit in the STSW is not set. If the processor detects a possible conflict, it then proceeds to check whether the OTID associated with the conflicting memory block is equal to the currently active OTID (by reading the STSW). If the IDs do not match, the processor proceeds without stalling or aborting (*i.e.*, the metadata is stale). If the IDs match, a conflict exists and the requesting processor stalls until the overflowed transaction clears the STSW's overflowed bit during commit. While a processor is stalling, another conflict can cause its transaction to abort.

If OTIDs were never reused, this approach would avoid the need to ever clear the metadata. However, the OTID width is finite and small. As a result, OTIDs will eventually wrap around, creating the potential for false conflicts and unnecessary delay. Such false conflicts can occur only when (1) an overflowed transaction is active (otherwise the metadata is ignored) and (2) the thread attempting the access is not executing in overflowed mode. The stall due to the false conflict will be temporary, because once the active overflowed transaction completes it clears the overflowed bit in the STSW, thus un-stalling the victim of the false conflict.

To reduce false stalls, the processor opportunistically clears stale overflowed transaction metadata whenever possible. Whenever a processor not executing an overflowed transaction writes a cache block, it clears the associated metadata. Thus, as long as a block has been written since the last time the current OTID was used, no false conflicts will occur on that block. The metadata can also be cleared whenever a processor manipulates a cache block in which the current OTID does not match the block's OTID. Lazily clearing metadata does not impact correctness or forward progress; it is only a performance optimization.

### 5.2.3 *Lazily Coherent Metadata*

Although the metadata can be kept exactly coherent by requiring a processor to have write permission to a block to modify its metadata, such a requirement causes unnecessary invalidations of blocks in shared state (and thus transaction conflicts) when only the metadata needs to be modified, and also inhibits the efficient lazy clearing of metadata.

Instead, we would like a processor executing an overflowed transaction to be able to set the metadata without needing exclusive permissions to the block. As there is only one active overflowed transaction at a time, there will be at most a single writer (even if there are multiple readable copies in the system). However, to prevent out-of-order writebacks from overwriting more recent metadata with stale metadata, the system allows only the owner of the block (non-exclusive or exclusive) to set the metadata. Many cache coherence protocols already include the notion of a single non-exclusive read-only dirty owner (the "O" state [28]) that is responsible for writing back the block to memory upon eviction. Once the metadata has been written, it is the owner's responsibility to ensure the data is eventually written back to memory (or transfer the ownership, and thus the responsibility, on to another processor). Some protocols already support a non-dirty owner as part of determining which processor responds with data during a shared-intervention [18, 29]. In protocols that grant non-exclusive owner status to the most recent requester, whenever a processor in an overflowed transaction requests a block, it will be able to set the read bit immediately after the miss completes.

The key to the correctness of this lazy updating of metadata is that the system guarantees that any *new* requests for the block receive the most recent version of the metadata. Once an overflowed transaction has set the read bit (and thus has the block in owned state), any other processor that tries to write the block will issue a cache request and receive the most recent version of the metadata, indicating the conflict. Processors will only set the write bit when they are writing the block, in which case they have exclusive permissions to it; thus, any subsequent read or write will again receive the most recent copy of the metadata and detect the conflict. Any processor can clear the metadata opportunistically as described above; if the processor owns the block, then its clearing of the metadata will propagate to other processors.

### 5.2.4 *Example Execution*

Figure 4 illustrates the lazy coherence and clearing of metadata. At time $t_1$, processor $P_1$ loads the block A into its cache; at that time, there is no overflowed transaction executing and the metadata for A is $\emptyset$. At $t_2$, $P_0$ overflows, setting the overflowed bit of the STSW and incrementing the OTID. At $t_3$, $P_0$ loads A into its cache in owned state and sets the read bit for A, as well as writing its OTID into the OTID metadata field for A. $P_1$ now has stale metadata in its cache, but there is no conflict. At $t_4$, $P_2$ loads A into its cache; because $P_0$ owns A, it supplies the data (and metadata) to $P_2$. Again, there is no conflict. At $t_5$, $P_3$ requests A in modified state; as the owner,

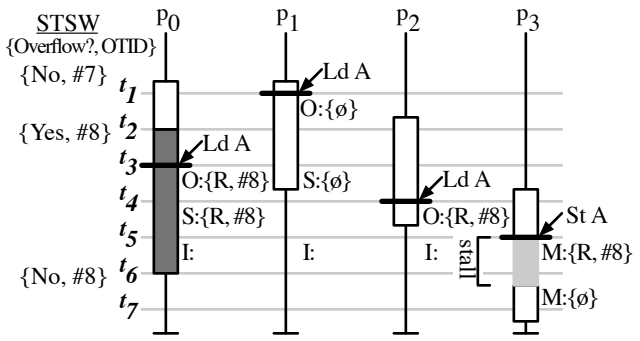


**Figure 4. Example illustrating lazy coherence and clearing of metadata in ONETM-Concurrent. The white bars are non-overflowed transactions, the dark gray bars are overflowed transactions, the straight lines are non-transactional execution, and the light gray color indicates stalled execution. The example centers around a memory block with address A; the text to the right of each processor is that processor's MOESI coherence state and local metadata for A.**

$P_2$ supplies $P_3$ with the data. $P_3$ now stalls, because the read bit of A is set and the STSW indicates that the OTID of the active overflowed transaction matches the OTID in the metadata of A. At $t_6$ $P_0$ commits its overflowed transaction, clearing the overflowed bit of the STSW. A short time later, $P_3$ sees that the overflowed bit of the STSW is now clear and unstalls itself to perform its write of A. It also opportunistically clears the metadata of A at this point. Because $P_3$ has A in modified state, it will ensure that its version of the metadata for A is given to anyone requesting A in the future.

## 5.3 Forward Progress

When a conflict occurs, a transaction must abort (or the request may be stalled [21]). The overflowed transaction is always the highest-priority transaction (*i.e.*, it will survive any conflict). A conflict between the overflowed transaction and a non-transactional access will stall the non-transactional access until the overflowed transaction commits. Thus, once a transaction has transitioned to overflowed mode, it will not abort due to conflicts (although it can abort at the explicit request of the software). If a transaction fails to make progress for any reason (*e.g.*, repeated aborts due to conflicts or interrupts), it arbitrates to become the overflowed transaction (thus ensuring forward progress).

## 5.4 Operating System Issues

On a context switch, a processor executing an overflowed transaction need not perform any special operations in either of our two implementations. In ONETM-Serialized, all other threads in the application will remain stalled until the overflowed transaction is swapped back in and completes. In ONETM-Concurrent, the metadata ensures that all other threads will detect conflicts with the overflowed transaction, even when it is not running.

Our designs require minimal support from the operating system. The operating system must save and restore the PTSW register as part of thread state. ONETM-Concurrent requires support for the overflowed metadata as well: when swapping pages to and from disk, the operating system must save and restore the associated metadata and OTIDs (as implemented in other systems [9, 27]). The operating system may optionally clear metadata and OTIDs when zeroing pages before reallocation.

## 5.5 Exposing the Highest Priority Transaction

Because an overflowed transaction is given highest priority in ONETM, after a transaction transitions into overflowed mode it will not abort unless explicitly instructed to do so by the software.

We can expose the concept of the highest-priority (overflowed) transaction to the software in two ways. First, if the compiler can guarantee through static analysis that the program will never explicitly ask for a transaction to be aborted (*e.g.*, via a language-level transactional abort), the software may mark the transaction as not needing support for explicit abort (for example, by setting a *no-user-abort* bit in the PTSW; see Figure 3b). If such a transaction transitions to overflowed mode, it can stop logging (and avoid the associated overheads) because it will never abort: neither due to conflicts (because overflowed transactions have highest conflict-resolution priority) nor software requests.

Second, although many common system calls and I/O may be handled via input/output buffering, compensation actions [4], and/or transactional OS interfaces, some operations are not easily handled within a transaction (*e.g.*, sending a network request and receiving its response). To handle cases in which a transaction wants to perform a non-transactional system call and can guarantee that a user abort will not occur, the runtime system can transition the transaction into overflowed mode with the no-user-abort bit set. As the transaction will then never abort, programs can rely on this property to, for example, perform arbitrary system calls or input/output within the transaction [3].

## 5.6 Qualitative Complexity Comparison

The complexity of ONETM-Concurrent compares favorably with that of the UTM, VTM and PTM proposals discussed in Section 4. To detect conflicts in the worst case, the three latter schemes walk a linked list of unbounded size. In ONETM-Concurrent, the conflict detection is direct and inexpensive, because the metadata bits travel with the data. On a commit of an unbounded transaction, ONETM merely clears the STSW bit. VTM traverses the linked list of updates that the transaction has made to propagate these updates to memory. UTM and PTM walk the linked list of updates in order to deallocate it. On aborts, all systems (including ONETM) walk the list of accesses made by the transaction. ONETM, however, walks a thread-local log (requiring no synchronization) whereas the other schemes walk a shared data structure.

The price that ONETM pays for the above simplifications is, of course, a limit on concurrency for overflowed transactions. In the previous section, we argued that the addition of the permissions-only cache can neutralize this cost. In the next section, we evaluate this claim quantitatively via full-system simulation.

# 6. Experimental Evaluation

This section evaluates the performance and scalability of our proposal. It examines both the performance impact of our concurrency restrictions on overflowed execution and of our permissions-only cache, finding the former to be small and the latter large.

## 6.1 Simulator and Benchmarks

We use Virtutech Simics [17] and a memory hierarchy simulator based on GEMS [19] that simulates our proposed transactional memory systems. The simulator models an in-order, one-IPC x86 processor and a bus-based memory hierarchy. Figure 5a presents the configuration parameters.

Our evaluation workload consists of a subset of the SPLASH-2 benchmark suite [30] and a binary tree microbenchmark. The benchmarks and inputs are summarized in Figure 5b. The *raytrace_opt* benchmark removes a frequently executed, but unnecessary, critical section from *raytrace*. We created transactional versions of these benchmarks by replacing lock acquires and releases with, respectively, transaction begins and ends.[3] The lock-based versions of the benchmarks use ticket locks handcrafted in x86 assembly to avoid the overhead of POSIX locks. We compiled the benchmarks using GCC 4.1.0 with the optimization flag -O3.

| Parameter | Value |
|---|---|
| Processor | eight in-order x86 cores, 1 IPC |
| L1 cache | 64 KB, 4-way set associative, 64B blocks |
| L1 miss latency | 10 cycles |
| L2 cache | 4 MB, 4-way set associative, 64B blocks |
| L2 miss latency | 200 cycles |

(a) Simulated machine configuration

| Program | Input |
|---|---|
| barnes | input-2K |
| cholesky | tk14.O |
| ocean-non-contiguous | n130 |
| radix | n262144 r1024 |
| raytrace | teapot.env |
| volrend | head-scaleddown2 |
| water-spatial | input-512 |
| tree-<*n*> | *n*% scanning |

(b) Benchmark summary

**Figure 5. Machine and benchmark summary.**

We use a binary tree microbenchmark to evaluate transactions with larger memory footprints than those found in SPLASH-2. The binary tree is 11 levels deep, and threads access the tree in a random mixture of (1) transactional lookup and single-node update and (2) transactional read scans of a randomly-selected contiguous subset of the values in the tree. This benchmark is parameterized by the division of execution time between these two operations. For example, in *tree-45*, 45% of uniprocessor execution is spent in read scans.

## 6.2 Performance

Figure 6 presents the execution time (normalized to a serial program) for the benchmarks on eight processors. For now, ignore the striped bars. The left-most bar (labeled "idealized") represents an idealized unbounded transactional memory system in which conflict detection is free and concurrency is unlimited for overflowed transactions. Any unbounded transactional memory strives to match the performance of this system. The light and dark gray bars represent our two proposed systems, ONETM-Concurrent and ONETM-Serialized, respectively. Finally, the black bar (labeled "locks") represents a lock-based implementation of each benchmark. Note that there are no lock-based bars for the *tree-n* benchmarks because they are so synchronization-heavy that performance is either overwhelmed by locking overhead (when fine-grained locks are used) or serialized (when coarse-grained locks are used).

In the SPLASH-2 benchmarks ONETM-Concurrent and ONETM-Serialized track the idealized unbounded transaction memory (the leftmost bar), because the transactions in these benchmarks require little overflowed execution. The *tree-n* benchmarks, on the other hand, do make use of overflowed execution. Despite the occurrence of overflowed transactions, for small

[3] We recognize that such a conversion from locks to transactions is not always safe [2].

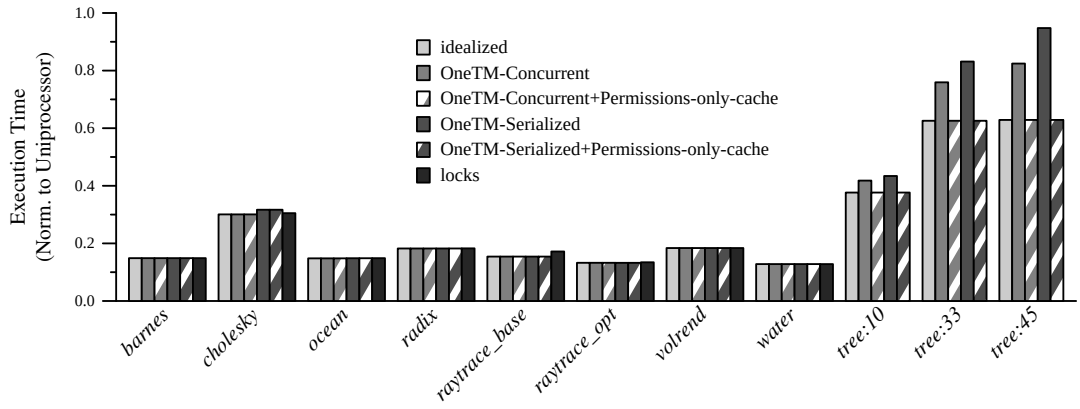


**Figure 6. Performance of ONETM implementations on eight processors versus an idealized, no-overhead unbounded transactional memory.**

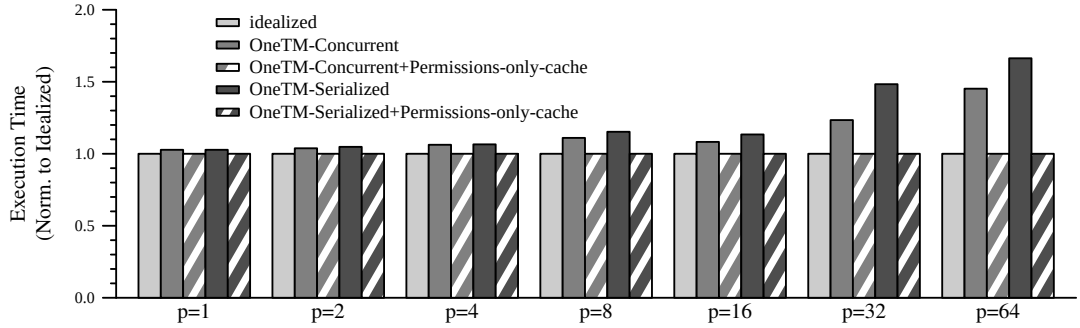


**Figure 7. Scalability analysis using the *tree-10%* microbenchmark.**

scanning percents (*i.e.*, *tree-10*), ONETM-Concurrent suffers little slowdown (versus the idealized system) and ONETM-Serialized suffers from a surprisingly modest slowdown. As the scanning percent grows, so does the likelihood that two threads require concurrent overflowed transaction, and both schemes suffer versus the idealized case. Naturally, ONETM-Concurrent degrades more slowly than ONETM-Serialized.

Next consider the striped bars in Figure 6. These bars represent ONETM-Concurrent and ONETM-Serialized with a permissions-only cache. We augment both systems with a 1KB permissions-only cache, allowing each system to reference up to 256KB before overflowed execution is necessary. Almost all of the overhead of ONETM-Concurrent and ONETM-Serialized is eliminated by the introduction of the permissions-only cache (*i.e.*, overflowed execution becomes sufficiently rare that the performance impact of the one-at-a-time limitation becomes negligible).

## 6.3 Scalability

To explore the scalability of the proposed implementations, Figure 7 shows the impact on performance of the *tree-10* benchmark as the number of processors in the system is varied. Execution times are normalized to the idealized case in each bar group. These results show that systems without the permissions-only cache suffer increasing overhead with larger numbers of processors, because a greater number of processors increases the likelihood that any given processor is executing an overflowed transaction at any given time. ONETM-Concurrent is more resilient to this effect because it does not serialize all threads in the benchmark, but it still exhibits significant slowdowns as processor count increases. However, the introduction of a permissions-only cache significantly reduces the rate of overflowed transactions, so the performance of both implementations is similar to the idealized implementation. Without the permissions-only cache, even the single-processor case exhibits slight slowdowns due to overheads of transitioning to overflowed execution mode.

## 6.4 Results Summary

These results show that allowing only one overflowed transaction at a time can result in performance competitive with an ideally concurrent implementation. For larger multiprocessors or processors with high-demand transactional workloads, however, the performance of ONETM-Concurrent suffers versus the ideal unbounded transactional memory. The addition of even a small permissions-only cache closes the gap for our benchmarks. Moreover, the permissions-only cache also equalizes the performance of ONETM-Serialized with that of the ideal in our experiments, suggesting that even serialization may be a viable implementation option when combined with the permissions-only cache.

# 7. Additional Related Work

This section describes some additional work in transactional memory as well as related work from the area of speculative synchronization.

LogTM-SE [31], a concurrently-developed proposal, and Bulk [5] use signatures for conflict detection rather than using different strategies for bounded and overflowed cases. A signature is a finite-length, conservative representation of a transaction's read and write sets that admits memory block aliasing. In LogTM-SE, conflict detection piggybacks on the cache coherence protocol: on an

incoming cache coherence request, the processor checks the signature to see if there is a conflict. The system summarizes the signatures of all swapped-out transactions in a single signature. One difference between signature-based approaches and both ONETM implementations is that signature-based approaches may incur false conflicts due to signature aliasing, but they allow full concurrency of overflowed transactions; in contrast, both ONETM implementations track conflicts exactly (albeit at a cache-block granularity) but limit applications to one overflowed transaction at a time. In addition, Bulk differs from our proposals in that it provides lazy (versus eager) conflict detection.

XTM [8] uses bounded transactions when possible and uses the virtual memory hardware to implement overflowed transactions. When a thread begins an overflowed transaction, it is given a private page table with restricted access so that the system can track transactional memory operations at the page granularity via exceptions. Memory updates cause shadow pages to be allocated, and conflicts are detected at commit time by checking that no other thread has updated a shadowed page. Our permissions-only cache could be employed to optimize the performance of XTM. One difference between XTM and ONETM is that XTM has an involved commit operation for overflowed transactions whereas ONETM has simple commit operations at the price of restricting applications to one overflowed transaction at a time.

Speculative Lock Elision [22] and Transactional Lock Removal [23] provide concurrent execution of lock-protected regions of a program by speculatively ignoring lock acquisition and executing optimistically (detecting conflicts and rolling back). These systems support transaction overflow and I/O by falling back on actually acquiring the lock associated with a locked region of code. This approach is similar to ONETM, albeit with a lock-based interface rather than the more powerful global serialization semantics of our proposal. A significant difference is that ONETM-Concurrent does not prevent non-overflowed transactions from making progress (unless there is an actual conflict). Our approach is also reminiscent of Speculative Synchronization's "safe thread" approach for ensuring forward progress [20].

# 8. Conclusion

Many unbounded transactional memory schemes separate transactional execution into a fast bounded mode and a slower overflowed mode. We introduced the permissions-only cache, which eases the memory-footprint restrictions imposed by the bounded execution mode of the processor. This structure permits bounded transactions to access potentially hundreds of megabytes (rather than tens of kilobytes) before overflowing. The permissions-only cache can be introduced into both hardware-only and hardware-software proposed systems to make the fast case more common.

We also proposed ONETM, an unbounded transactional memory system that assumes overflowed transactions will be rare to simplify the implementation. Specifically, we limit the number of overflowed transactions that may exist in an application at a time to one. By bounding concurrency among overflowed transactions, ONETM avoids the complexity of managing and traversing linked data structures in hardware, admitting simple conflict detection and transaction commit.

Our experimental evaluation demonstrates that the permissions-only cache can significantly reduce the use of overflowed execution and that ONETM can obtain performance that is comparable to an idealized unbounded transactional memory system. Our results indicate that the combination of the permissions-only cache and the simplified overflowed execution mode of ONETM represents a practical design point for unbounded transactional memory systems.

# Acknowledgments

The authors thank Mark Hill, Kevin Moore, Amir Roth, and Dan Sorin for comments on this work. We thank Dan Sorin and Albert Meixner for discussions about their token-only cache idea. This work is supported in part by two NSF CAREER awards (CCF-0644197 and CCF-0347290), National Science Foundation grant CCF-0311199, and donations from Intel Corporation.

# References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE TCCA Computer Architecture Letters*, 5(2), Nov. 2006.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr. 2006.

[4] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[5] L. Ceze, J. M. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[6] Y. Chou, L. Spracklen, and S. G. Abraham. Store Memory-Level Parallelism Optimizations for Commercial Applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, Nov. 2005.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded Page-Based Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, Oct. 2006.

[8] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 371–381, Oct. 2006.

[9] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, Oct. 2006.

[11] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Archi-*

*tectural Support for Programming Languages and Operating Systems*, pages 1–13, Oct. 2004.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Mar. 2006.

[15] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[16] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[17] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[18] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[20] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[22] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[23] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[24] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, June 2005.

[25] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[26] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. S. III, , and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[27] F. G. Soltis. *Inside the AS/400*. Duke Press, 2nd edition, 1997.

[28] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[29] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[31] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.